V.06.

Общий обзор занятия.

Ещё раз говорим про линейные структуры. И опять начинаем, разумеется, с задачи. И задача сегодня одна, зато какая! Сортировка. Вечная, многогранная, чрезвычайно разнообразная. И сегодня мы не в первый раз к ней прикасаемся, и ещё будем смотреть на неё с различных позиций, причём уже в этом семестре. Но сегодня мы, кажется, впервые прикоснёмся к ней так, регулярным образом, хотя сегодня у нас совсем простые алгоритмы. Да, алгоритмы. И сегодняшние алгоритмы организуют на сортируемые объекты (мы будем работать с числами, но понятно, что совершенно неважно, что именно сортировать), в линейную структуру, в этакий отрезок с данными.

Маленькое отступление-напоминание. Как и большинство линейных структур (линейных в смысле структур, где данные выстроены в линию, не разветвляются, где узлы в любой момент можно просто пронумеровать по порядку) линейные структуры можно реализовать на слайсе, но уже в случае с задачей Иосифа, где надо вычёркивать узлы из середины списка, а не с краёв, связные списки вполне могут ускорить обработку. Сегодня нам понадобится вставлять новые узлы в середину списка, что тоже на списках делается мгновенно, а на слайсах запросто может и встормозить. Но решения на слайсах тоже рассмотрим, хотя и без особого обсуждения - дети вполне уже сами могут разобраться, пусть и разбираются. Конец отступления, переходим в наступление.

И на этом фоне продолжим свою линию - строить структуры данных, как можно более адекватные выбранному алгоритму решения задачи. Но тут уже, пусть и не очень громко, но проявится и обратная сторона - выбор структуры данных вличет обратно на алгоритм обработки. Да, эти два момента (полностью исчерпывающие весь процесс поиска решения) довольно естественно находятся в этаком диалектическом взаимодействии, не просто алгоритм вытягивает на себя соответствующие структуры данных, а они влияют друг на друга, изменяя, подправляя, уточняя друг друга. В общем-то, совершенно естественное поведение, но у нас пока это не очень звучало, в основном впереди шел алгоритм, а потом уже структура данных. Тоже довольно естественно - ведь мы пока только начали учиться обращаться со структурами, нам бы набрать хоть какой-то опыт работы с ними, какие-то технические навыки, как-то начать жить с

ними. Ну, хочется надеяться, что этот самый первоначальный момент мы уже прошли. Будем развивать свои представления о структурах данных, попробуем разглядеть (ну, хоть начать), что выбор представления данных влияет обратно на алгоритм, а тот обратно на представление и т.д., пока не устаканимся на чём-нибудь хорошем, а, хорошо бы, и интересном.

Конечно, технический момент, развитие технических навыков, понимания и наработка техники - всё равно сейчас важнейшая задача, но говорить про неё особо больше нечего. Пока.

Да, и под занавес немного поговорим о `package Container` - это как раз некоторые популярные структуры данных, контейнеры для хранения конкретных данных, реализованные библиотечно. В этой библиотеке реализованы куча (о которой в это раз мы не будем говорить вообще никак, но к которой обязательно придём, когда речь пойдёт о рекурсии в применении к алгоритмам обработки структур данных) замечание: эта куча *(heap)* - это совсем не та куча, в которой выделялась память динамически, но что поделаешь, такая терминология сложилась...; реализованы также двусвязный список и кольцо (циклический список), о двусвязных списках поговорим.

И ещё раз повторюсь: каждый раз обращаем внимание явно на связь АТД <-> реализация, т.е. набор операций, набор способов доступа к данным <-> конкретная реализация имеющимися средствами конкретного средства программирвания (языка `Go` в нашем случае).

И снова у нас есть и принципиальный момент - конструирование АТД, и технический момент - реализация. И опять, пока дети ещё совсем не привыкли, пока у них ещё нет уверенного навыка работы со связными структурами, техническому моменту надо уделять достаточно много внимания, особенно на практике. Пожалуй даже больше, чем принципиальному. Техника сейчас крайне важна. Достигается техника только упражнениями. Не скупитесь :) Упражнения на списки (список упражнений был в прошлый раз) - хороший материал, но фантазитровать тут можно как угодно много.

Ну, поехали.

Лекция

Сортировка вставками.

Говорить сегодня мы будем о сортировках вставками. Их много разных есть, мы рассмотрим только парочку очень простеньких, ведь наша цель сегодня не построить суперэффективную сортировку, а поговорить о структурах даннх. Суть всех сортировок вставками такова: берём сортируемые объекты по одному, к тому моменту, когда дело доходит до K-го объекта, предыдущие (K-1) объект уже отсортирован; находим среди них место K-го и вставляем его на это полагающееся ему место.

Метод простых вставок.

Простейший метод сортировки посредством вставок совсем очевиден. Сравниваем по очереди новый элемент с уже упорядоченными, двигаясь от наименьшего к наибольшему, до тех пор, пока он больше прежних элементов. Как только мы обнаружим, что новый элемент меньше (или равен) какого-то элемента, скажем для определённости, меньше либо равен элемента, стоящего на месте номер J, так мы сдвигаем вправо все элементы, начиная с J-го, а новый элемент ставим на место номер J. Если совсем коротко: каждый раз бежим по уже упорядоченному ряду, пока не найдём, куда вставлять очередной элемент.

Давайте уже начинать конкретизироваться. Как ни приятно говорить об алгоритме в максимально общей постановке, но нам надо рассмаривать примеры, так что давайте договоримся, что сортировать будем целые числа - int'ы, хотя это вообще не играет никакой роли.

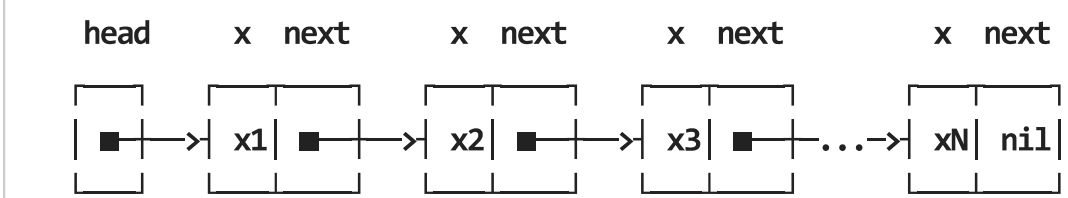
Совершенно очевидно здесь напрашиваются две структуры данных для хранения наших чисел: слайс и связный список. Различий в алгоритмах для этих двух представлений данных совсем немного, но давайте рассмторим их отдельно. И начнём со связного списка.

Алгоритм сортировки простыми вставками для связного списка.

Хранить данные будем точно так, как это было у нас в прошлый раз:

```
type list struct {
    head *lmnt
}
```

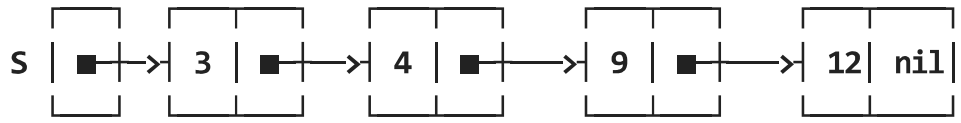
```
type lmnt struct {
    x int
    next *lmnt
}
```



Элементы списка - типа lmnt; x1, x2 ... - сортируемые числа.

Бегать по списку надо только в одну сторону - вперёд, так что ограничимся однонаправленным (а не двусвязным) списком.

И начнём с самого штатного случая (пограничные случаи оставим пока). Вот мы имеем ситуацию:



Четыре числа - 3, 4, 9 и 12 - уже отсортированы, пытаемся вставить число 7, которое хранптся в переменной num.

Быстро-быстро начинаем набрасывать код типа такого

```
runner:= s.head
for (*runner).x < num  { runner = (*runner).next }
```

и резко обламываемся: остановка происходит на том числе, перед которым надо вставлять, а у нас указатели назад не смотрят. Ага, значит надо заглядывать вперёд - переписываем код:

```
runner:= s.head
for (*(*runner).next).x < num  { runner = (*runner).next }
```

Можно также рассмотреть вариант с двумя бегущими указателями – один (runner) отстаёт от другого (runner2) на шаг:

```
var runner, runner2 *lmnt
runner2 = s.head
for (*runner2).x < num  {
  runner, runner2 = runner2, (*runner2).next
}
```

И тогда в обоих вариантах вставляем новое число вслед за runner'ом:

```
(*runner).next = &lmnt{num, (*runner).next}
```

Здесь обязательно это место прорисовать, желательно в двух цветах – что есть, что надо, - и в динамике, как что изменяется.

И вроде получили простенькую и элегантную функцию. Но не тут-то было...
Все неприятности происходят не в полёте, все неприятности происходят при взлёте и посадке. Пограничные случаи!

Какие здесь возможны пограничные случаи? На границах, естественно.

1. Новое число больше последнего (максимального) числа.
   Приходится менять код поиска. Получим что-то вроде:

   ```
   runner:= s.head
   for ((*runner).next!=nil) && ((*(*runner).next).x < num)  {
     runner = (*runner).next
   ```

```
}
```

или, в варианте с двумя бегущими указателями:

```
var runner, runner2 *lmnt
runner2 = s.head
for (runner2 != nil) && ((*runner2).x < num)  {
runner, runner2 = runner2, (*runner2).next
}
```

Вставка нового элемента в список осталась той же:

```
(*runner).next = &lmnt{num, (*runner).next}
```

Но каждая проверка теперь выполнят двойную работу. Да и код утяжелился. Но это ещё не всё. Ведь возможно ещё, что

2. Новое число не превосходит первого (минимального)
   Да. Придётся перед циклом поиска поставить соответствующую проверку.

   ```
   if num <= (*runner).x ...
   ```

   или в варианте с двумя указателями

   ```
   if num <= (*runner2).x ...
   ```

   Код утяжеляется прямо на глазах. Но и это ещё не всё. Есть ещё одна пограничная ситуация – не с чем сравнивать новое число.

3. Список пуст

   ```
   if (*s).head == nil ...
   ```

   Да, конечно, такое бывает один раз на весь список. Но, тем не менее, такая проверка должна выполняться каждый раз при добавлении в список нового числа.

Печаль. От былой элегантности кода остались жалкие брызги...

Вот что у нас в получилось:

программа linkedlist1.go.

```
// linkedlist1.go
package main

import     (
    "fmt"
)

type  (
    list struct {
        head *lmnt
    }
    lmnt struct {
        x int
        next *lmnt
    }
)

func initList() list  {
    return list{nil}
}

func (s list) Print()  {
    for runner:= s.head; runner != nil; runner = (runner).next  {
        fmt.Print((*runner).x, " ")
    }
    fmt.Println()
}

func (s *list) Insert1(num int)  {
    runner:= (*s).head
    if runner == nil  {
        (*s).head = &lmnt{num, nil}
        return
    }
```

```
    if num <= (*runner).x {
        (*s).head = &lmnt{num, runner}
        return
    }
    for ((*runner).next!=nil) && ((*(*runner).next).x < num)  {
        runner = (*runner).next
    }
    (*runner).next = &lmnt{num, (*runner).next}
}

func (s *list) Insert2(num int)  {
    var runner, runner2 *lmnt
    runner2 = (*s).head
    if runner2 == nil  {
        (*s).head = &lmnt{num, nil}
        return
    }
    if num <= (*runner2).x {
        (*s).head = &lmnt{num, runner2}
        return
    }
    for (runner2 != nil) && ((*runner2).x < num)  {
      runner, runner2 = runner2, (*runner2).next
    }
    (*runner).next = &lmnt{num, (*runner).next}
}

func main() {
    l:= initList()
    l.Insert2(2)
    l.Insert2(5)
    l.Insert2(4)
    l.Insert2(1)
    l.Insert2(2)
    l.Insert2(8)
    l.Print()
```

```
}
```

В этой программе сведены вместе функции `Insert1` и `Insert2`: первая реализует вариант с “заглядыванием вперёд”, вторая - вариант с двумя синхронно двигающимися указателями - на текукщий элемент и на предыдущий. Они совершенно не отличаются по функциональности, использовать можно любую из них, в одном коде они живут исключительно ради экономии места, да и понимать так полегче.

И вот тут-то мы и думаем: все неприятности случались в начале и конце списка – так не избавиться ли от них одним махом?! Да, избавиться – сделать так, чтобы список никогда не был пустым и не случались случаи 1 (вставка нового числа в конец имеющегося списка) и 2 (вставка нового числа в начало имеющегося списка). Естественно, для этого надо присобачить к списку два элемента: в начало – минус бесконечность, в конец – плюс бесконечность. А что такое бесконечвность? Туповатый вопрос, на первый взгляд. Это на первый, пока не встанет вопрос о реализации. Да, необходимость реализации сильно проветривает затхлость в голове. Так что же такое бесконечность? И, как это часто бывает, надо только задуматься, и тогда всё становиться ясным и несложным. Бесконечность - это то,что больше всего. Та величина, которая больше любой возможной величины. Да, фиктивная, но от этого не менее реалистичная. И, понятное дело, минус бесконечность - это то, что меньше всех возможных значений. Так что актуальная бесконечность - штука переменчивая, зависит от контекста. Если у нас речь идёт о датах рождения, то уместно взять 1.1.1000 и 1.1.3000; если об исторических датах – то соответствующие даты; если речь идёт о фамилиях – берём фамилии ‘A’ и ‘ZZZZZZZZZZZZZ’, ну и т.д. И все проблемы уходят разом! Да, мы за это платим памятью под два элемента списка, но, не говоря уже о том, что скорость работы процедуры увеличивается (и заметно), так мы ещё и экономим память. память экономим за счёт уменьшения объёма кода. А о читабельности, простоте и ясности кода нечего и говорить.
Вот он, получившийся код:
программа linkedlist2.go.

```
// linkedlist2.go
package main

import     (
```

```
    "fmt"
    "math"
)

const  (
    PlusInfinity = math.MaxInt64
    MinusInfinity = math.MinInt64
)

type  (
    list struct {
        head *lmnt
    }
    lmnt struct {
        x int
        next *lmnt
    }
)

func initList() list  {
    return list{&lmnt{MinusInfinity, &lmnt{PlusInfinity, nil}}}
}

func (s list) Print()  {
    for runner:= (*s.head).next; (*runner).x < PlusInfinity; runner = (runner).n
ext  {
        fmt.Print((*runner).x, " ")
    }
    fmt.Println()
}

func (s *list) Insert1(num int)  {
    runner:= (*s).head
    for (*(*runner).next).x < num  {
        runner = (*runner).next
    }
```

```
    (*runner).next = &lmnt{num, (*runner).next}
}

func (s *list) Insert2(num int)  {
    runner:= (*s).head
    runner2:= (*runner).next
    for (*runner2).x < num  {
        runner, runner2 = runner2, (*runner2).next
    }
    (*runner).next = &lmnt{num, runner2}
}

func main() {
    l:= initList()
    l.Insert2(2)
    l.Insert2(5)
    l.Insert2(4)
    l.Insert2(1)
    l.Insert2(2)
    l.Insert2(8)
    l.Print()
}
```

Вот так вот, начали мы с задачи, решили хранить это дело в списке, а потом довели список до ума – получили вот такую вот структуру "список с навесками", который очень хорошо прикладывается к задаче.

Алгоритм сортировки простыми вставками для слайса.

Алгоритм есть - сортировка простыми вставками. Кратко описывается он, повторю, как-то так: каждый раз бежим по уже упорядоченному ряду, пока не найдём, куда вставлять очередной элемент.

Что изменилось с заменой связного списка на слайс? Появилась возможность вставлять новый элемент перед каким-то определённым элементом. Отпадает особый случай со вставкой нового числа перед первым числом в списке - случай, когда новое число не превосходит текущего минимального числа. Что остаётся?

Остаются случаи

- новое число больше последнего (максимального) числа, и
- список пуст
  Оба этих особых случая “убиваются” добавкой одного фиктивного элемента - элемента `+бесконечность` .

В итоге получаем код:
программа slice.go.

```
// slice.go
package main

import     (
    "fmt"
    "math"
)

const
    Infinity = math.MaxInt64

type
    list []int

func initList() list  {
    return list{Infinity}
}

func (s list) Print()  {
    for i, x:= range s  {
        if i == len(s) -1  { break }
        fmt.Print(x, " ")
    }
    fmt.Println()
}

func (s *list) Insert(num int)  {
```

```
    for i, x:= range *s  {
        if x >= num  {
            *s = append(*s, 0)
            copy((*s)[i+1:], (*s)[i:])
            (*s)[i] = num
            break
        }
    }
}

func main() {
    l:= initList()
    l.Insert(2)
    l.Insert(5)
    l.Insert(4)
    l.Insert(1)
    l.Insert(2)
    l.Insert(8)
    l.Print()
}
```

package container/list. Библиотечный двусвязный список.

Поставлю сразу сюда линк на Описание package container/list и заодно здесь поставлю краткую выларжку: только общий обзор и заголовки всех методов типа list.List

Package list

```
import "container/list"
```

Обзор

Package list реализует двусвязный список (doubly linked list).

Пройтись по списку можно так (здесь l это *List):

```
for e := l.Front(); e != nil; e = e.Next() {
    // обрабатываем e.Value
}
```

Index

Заголовки методов типов list.Element и list.List совершенно понятны и трактуются, imho, однозначно и без проблем, так что заголовками и ограничимся. Подробное описание с нюансами - по вышеприведённой ссылке, там есть ответы на всякие вопросы типа “а что будет, если мы хотим узнать первый элемент пустого списка?” или “а что будет, если мы хотим удалить элемент, которого нет в списке?”, хотя они (ответы) вполне себе естественны и предсказуемы.

```
type Element
    func (e *Element) Next() *Element
    func (e *Element) Prev() *Element
type List
    func New() *List
    func (l *List) Back() *Element
    func (l *List) Front() *Element
    func (l *List) Init() *List
    func (l *List) InsertAfter(v interface{}, mark *Element) *Element
    func (l *List) InsertBefore(v interface{}, mark *Element) *Element
    func (l *List) Len() int
    func (l *List) MoveAfter(e, mark *Element)
    func (l *List) MoveBefore(e, mark *Element)
    func (l *List) MoveToBack(e *Element)
    func (l *List) MoveToFront(e *Element)
    func (l *List) PushBack(v interface{}) *Element
    func (l *List) PushBackList(other *List)
    func (l *List) PushFront(v interface{}) *Element
    func (l *List) PushFrontList(other *List)
    func (l *List) Remove(e *Element) interface{}
```

Но есть один совсем непростой вопрос: а что такое `interface{}` . И об этом мы в этом семестре говорить не будем, будем в следующем. А сегодня скажем примерно следующее. Список может быть любого типа, в смысле элементы списка могут быть любого типа, желательно, конечно, одного и того же, но даже и это не обязательно. так вот, скажем, что пока можем воспринимать interface{} как произвольный тип. Но чтобы исполнять с переменными типа interface{} какие-то действия нам же надо знать конкретный тип - ведь с разными типами данных можно исполнять разные операции. Так вот, чтобы указать явно конкретный тип мы используем так назыаваемый type assertion. Просто скажем - делай так: если

```
var i interfsce {}
```

то

```
i.(конкретный тип)
```

трактует `i` именно как величину этого конкретного типа, даёт доступ к базовому конкретному значению величины интерфейса. Более того,это выражение утверждает, что интерфейс i содержит именно это конкретный тип. А если не содержит или содержит не то? Если это утверждение не соответствует действительности? Понятно, тогда panic. Можно, и тут синтаксис очень похож на синтаксис мапов, проверить, допустим ли такой assertion (собственно, в данном контексте assertion точнее всего и преводиится словом “утверждение”, т.е. верно ли, корректно ли такое утверждение) - двухаргументный assertion.

Короче и проще тупо приведу пример из A Tour of Go - он всё объясняет.

```
package main

import "fmt"

func main() {
    var i interface{} = "hello"

    s := i.(string)
    fmt.Println(s)
```

```
    s, ok := i.(string)
    fmt.Println(s, ok)

    f, ok := i.(float64)
    fmt.Println(f, ok)

    f = i.(float64) // panic
    fmt.Println(f)
}
```

И ещё один пример, в котором реализована всё та же сортировка простыми вставками именно на списке типа list.List: **программа containerlist.go.**

```
// containerlist.go
package main

import     (
    "container/list"
    "fmt"
    "math"
)

const  Infinity = math.MaxInt64

func main() {
    // Create a new list and put some Infinity in it.
    l := list.New()
    l.PushFront(Infinity)

    for i:= 3; i>0; i = (i+6)%19 {
        fmt.Print(i, " ")
        for e := l.Front(); ; e = e.Next() {
            if e.Value.(int) >= i  {
                l.InsertBefore(i, e)
                break
            }
```

```
        }
    }
    fmt.Println()
    l.Remove(l.Back())

    // Iterate through list and print its contents.
    for e := l.Front(); e != nil; e = e.Next() {
        fmt.Print(e.Value, " ")
    }
    fmt.Println()
//  Output:
//  3 9 15 2 8 14 1 7 13
//  1 2 3 7 8 9 13 14 15
}
```

Сравниваем варианты сортировок по скорости. Benchmark.

Бенчмарчить здесь и важно, и интересно. И кажется, что дети не впорлне осознали, как именно это делать. А здесь как раз очень подходящее место и время для смравнения реализаций. Ниже приводятся результаты шести сортировок.

```
- list1_1. Список без бесконечостей, пробегаем, заглядывая вперёд
- list1_2. Список без бесконечостей, пробегаем с двумя указателями
- list2_1. Список с бесконечостями, пробегаем, заглядывая вперёд
- list2_2. Список с бесконечостями, пробегаем с двумя указателями
- slice. Реализация списка на слайсе
- container/list. Библиотечный двусвязный список
```

```
          list1_1  list1_2  list2_1  list2_2    slice  container/list
Sort256     46520    47195    48281    50042    32307    97751     (ns/op)
Sort512    160276   164618   168254   165193   142712   215204     (ns/op)
Sort1024   561822   595194   577762   620535   333253   962729     (ns/op)
```

```
Sort2048    2250187   2274455   2356868   2321832   1194813   5114139      (ns/op)
```

Сами тексты программ я сюда не включаю - там все те же сортировки, только упакованные в benchmark, но детям их надо дать обязательно - этот навык нужен. Пусть отрабатывают. Все **тексты лежат в каталоге ./samples/benchmark**

Немного странные результаты. Но дело в том, что объёмы сортируемых данных очень маленькие - 256/512/1024/2048 элементов. А много и нельзя, сортировка простыми вставками - весьма медленная. А на малых объемах, на малых размерах вставка в слайс происходит сравнительно быстро. Тем не менее, результаты интересные.

Последний вопрос: а как сортировать бОльшие объёмы? А это другая история, о которой мы ещё поговорим обязательно в этом семестре.

Задания и упражнения.

Упражнений чегодня новых не видно. Надо делать упражнения на списки, в материалах к прошлому занятию их полно.

А задания есть.

Простые обменные сортировки: “пузырь” и “шейкер”

Сортировки широко известные, так что конкретных линков не будет - их полно, выбрать трудно, в конце концов можно просто посмотреть в Wiki, так что только кратко опишу суть алгоритмов и чуть-чуть прокомментирую некоторые технические моменты.

В простейшем варианте суть обменных сортировок излагается просто.

```
Сначала у нас есть неотсортированный список и мы  многократно проходим по списку . Один проход состоит в следующем: бежим по списку от начала к концу и на каждом  шаге сравниваем текущиё элемент списка и следующий; если необходимо, меняем пор ядок их следования в списке. Заканчиваем процесс, когда в течение очередного про хода по списку мы не совершим ни одного обмена.
```

Несколько технических замечаний.

1. Реализовывать можно. конечно и на слайсе, но крайне желательно делать это на связном списке - это очень хорошее задание на связные списки
2. Порядок следования соседних элементов меняем только за счёт изменения значений указателей, конечно, никак иначе. И этот момент можно использовать, чтобы показать ещё раз технику работы с указателями – двуцветные рисунки, динамика процесса и т.д. по полной программе.
3. Видимо, в этой задаче стоит сразу начать со списка с плюс-минус бесконченостями, потому что здесь без плюс-минус бесконечностей получается совсем гадко.

Это был беспонтовый и совершенно не применяемый «пузырь», он же “bubblesort”.

А вот если мы каждый раз будем изменять направление обхода списка: сначала слева направо, потом справа налево, потом опять слева направо, потом снова справа налево и т.д., то у нас получится вполне востребованная “шейкерная сортировка”, она же “coctail sort”, она же “двунаправленная пузырьковая сортировка”. Она весьма эффективна и часто применима для “почти отсортированных” списков.

Совершенно ясно, что, поскольку бегать придётся туда-сюда, то здесь необходим двусвязный список (мы о нём говорили в прошлый раз в связи с реализайцией дека), коорый мы держим, естественно, за оба конца - и за левый, и за правый.

И, да, опять-таки, можно это реализовывать на слайсе, но давайте делать это на двусвязном списке, именно на вот таком: